

## RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

### Techniques d'analyse de programmes pour la rétro-ingénierie de bases de données

Henrard, Jean; Hick, Jean-Marc; Roland, Didier; Englebert, Vincent; Hainaut, Jean-Luc

*Published in:*  
INFORSID'96

*Publication date:*  
1996

#### [Link to publication](#)

*Citation for pulished version (HARVARD):*

Henrard, J, Hick, J-M, Roland, D, Englebert, V & Hainaut, J-L 1996, Techniques d'analyse de programmes pour la rétro-ingénierie de bases de données. Dans *INFORSID'96*. INFORSID, p. 215-232.  
<<http://www.info.fundp.ac.be/cgi-bin/pub-spec-paper?RP-96-018>>

#### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

#### Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

# Technique d'analyse de programmes pour la rétro-ingénierie de bases de données

**J. Henrard, J.-M. Hick, D. Roland, V. Englebert, J.-L. Hainaut**

Institut d'Informatique, 21 rue Grandgagnage, 5000 Namur, Belgique  
Tél: 32 81 72 49 85, Fax : 32 81 72 49 67  
E-mail : db-main@info.fundp.ac.be

---

## Résumé :

Cet article propose d'appliquer une méthode générique de rétro-ingénierie de bases de données à une étude de cas. Nous présenterons brièvement une méthode générique de rétro-ingénierie de bases de données. Ensuite, nous décrirons l'atelier logiciel DB-MAIN et ses fonctionnalités utilisées lors de la rétro-ingénierie. Et plus particulièrement la notion de fragmentation de programme, qui est une technique puissante et efficace de compréhension du comportement d'un programme en un point donné. Le tout sera mis en oeuvre dans une étude de cas de complexité réaliste bien que de taille limitée.

**Mots clés :** rétro-ingénierie, fragment de programme, DB-MAIN, base de données, méthode

---

## Abstract :

In this paper we will apply a generic database reverse engineering methodology to a case study. We will sketch a database reverse engineering methodology. Then, we will describe the DB-MAIN CASE tool and its reverse engineering functionality. We will explain more precisely the program slicing. This is a powerful and useful technique to understand a program at a given point. All will be put together in a realistic, but small, case study.

**Key words :** reverse engineering, program slicing, DB-MAIN, database, method

---

## 1. Introduction

La rétro-ingénierie d'un composant logiciel est un processus d'analyse de la version opérationnelle de ce composant qui vise à en reconstruire les spécifications techniques et fonctionnelles. La rétro-ingénierie a comme but la redocumentation, la conversion, la maintenance ou l'évolution d'anciennes applications.

La rétro-ingénierie est un processus d'autant plus complexe que l'application est mal structurée, ancienne, non ou mal documentée. Pour les applications orientées données, c'est-à-dire les applications dont le point central est une base de données, la complexité peut être réduite en considérant que l'on peut effectuer la rétro-ingénierie des bases de données indépendamment (ou presque) de la partie procédurale. Cette approche peut se justifier de la façon suivante :

- les données persistantes constituent le composant central de nombreuses applications de gestion;
- la connaissance de la structure des données persistantes facilite la compréhension de l'application complète;
- la distance sémantique entre la spécification conceptuelle et l'implémentation physique est souvent plus faible pour les données que pour le code procédural;
- la structure des données persistantes est généralement la partie la plus stable d'une application;
- la méthodologie des bases de données est plus formalisée que celle du logiciel en général.

La rétro-ingénierie des structures de données bien qu'étant mieux maîtrisée que celle d'applications complètes, demeure une tâche complexe. La majorité des propositions de méthodes imposent des hypothèses trop restrictives pour traiter complètement des applications complexes :

- la base de données a été obtenue via des règles de transformation conceptuel/logique simplistes, et par conséquent la traduction du schéma physique en schéma conceptuel est presque immédiate;
- le schéma n'a pas subi de restructuration d'optimisation;
- toutes les contraintes ont été traduites dans le langage de description de données;
- les noms sont significatifs;
- les méthodes sont spécifiques à un type de SGBD<sup>1</sup>;
- pas (ou peu) de prise en compte du code procédural.

La plupart des applications réelles violent ces hypothèses, ce qui rend leur rétro-ingénierie plus complexe. Elle ne peut être menée à bien sans l'aide d'outils puissants et adaptés.

Après avoir décrit une démarche méthodologique générique, indépendante du SGBD, et l'outil DB-MAIN, nous allons appliquer cette méthode à la résolution d'un problème de complexité réelle, mais de taille limitée.

Cet article est organisé comme suit. La section 2 est une synthèse la méthode générique de rétro-ingénierie de bases de données. La section 3 présente la notion de fragmentation de programme qui sera utilisée pour l'analyse du code source. La section 4.1 décrit les principaux concepts et les fonctionnalités de l'atelier logiciel DB-MAIN. Dans la section 4.2, les fonctionnalités utilisées pour

---

<sup>1</sup>System de Gestion de Bases de Données

la rétro-ingénierie seront brièvement présentées. Une étude de cas complète sera développée dans la section 5.

## 2. Une méthode générique de rétro-ingénierie de bases de données

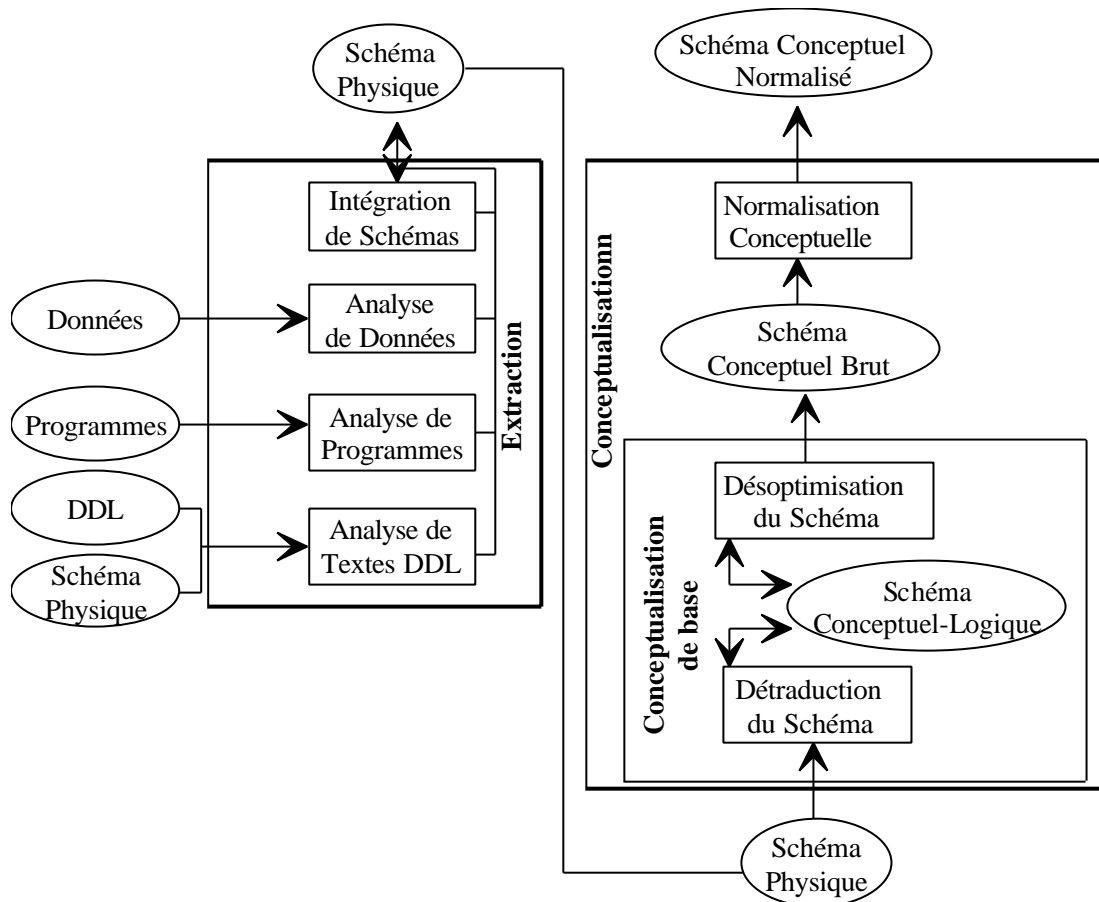


Figure 1 : Méthode générique de rétro-ingénierie de bases de données

Cette méthode comporte deux processus principaux (figure 1), à savoir l'extraction des structures de données et la conceptualisation de ces structures [Hainaut et al. 95]. Ces deux parties correspondent à des schémas différents qui requièrent des raisonnements, des concepts et des outils différents. De plus, cette division est à peu près l'inverse des processus de conception physique et logique habituellement admis en ingénierie des bases de données.

### 2.1. L'extraction des structures de données

Cette phase a pour objet la reconstruction du schéma logique complet selon le modèle du SGD<sup>2</sup>, y compris toutes les contraintes et les structures de données non explicitement déclarées.

Certains SGD (les SGBD par exemple) offrent, sous une forme ou une autre, une description du schéma global des données. Bien que ce schéma soit déjà assez complet, il devra être enrichi au moyen d'une analyse des autres composants de l'application (vues, code procédural, données, écrans de saisie, ...).

<sup>2</sup>System de Gestion de Données

Le problème est beaucoup plus complexe quand il s'agit de recouvrer le schéma conceptuel de fichiers classiques. Chaque programme source devra être analysé pour retrouver une partie de la structure des données. Cette analyse doit aller bien au-delà de la simple recherche de la structure des fichiers déclarés dans les programmes.

En particulier, deux types de problèmes peuvent être rencontrés, quelque soit le SGD : les structures sont définies dans les programmes ou il y a perte de spécifications. Il est possible que lors de la conception de l'application l'on n'ait pas utilisé toutes les possibilités du SGD et implémenté dans le programme certaines contraintes qui auraient pu être déclarées dans le SGD.

Les *structures définies dans les programmes* sont des structures ou des contraintes qui ne sont pas déclarées dans le SGD mais qui sont représentées ou vérifiées de façon procédurale dans le programme. Par exemple : des contraintes référentielles.

La *perte de spécifications* vient de la non-implémentation dans le SGD et dans le programme, de certaines contraintes du schéma conceptuel. Elles peuvent, par exemple, être vérifiées par construction : les données sont importées et sont correctes; l'exécution du programme ne permet pas de violer ces contraintes.

Recouvrer les structures définies dans les programmes ou perdues est une tâche complexe, pour laquelle il n'existe pas encore de méthodes déterministes. Seule une analyse méticuleuse de toutes les sources d'informations disponibles permet de retrouver les spécifications. Le plus souvent ces informations doivent être consolidées par la connaissance du domaine de l'application.

Les processus principaux de l'extraction des structures de données sont les suivants :

- analyse des déclarations des structures de données dans les scripts de définition du schéma et dans les sources du programme;
- analyse du code source du programme pour recouvrer les structures définies dans les programmes;
- analyse des données sur lesquelles travaille le programme pour en trouver les structures et les propriétés ainsi que pour confirmer ou infirmer certaines hypothèses;
- intégration des différents schémas obtenus lors des étapes précédentes.

Dans cet article, nous tenterons d'illustrer une technique particulièrement puissante d'analyse de programmes qui permet de détecter des structures algorithmiques qui suggèrent des structures de données et des contraintes.

## **2.2. La conceptualisation des structures de données**

Le deuxième processus consiste en l'interprétation du schéma obtenu lors de l'extraction des structures de données pour en dériver un schéma conceptuel. Il détecte et transforme (ou élimine) les redondances et les structures non conceptuelles introduites lors de la conception de la base de données. La conceptualisation des structures de données se fait en trois étapes : la préparation du schéma, la conceptualisation de base et la normalisation.

La *préparation du schéma* consiste en la modification de certains noms de manière à les rendre plus significatifs.

Lors de la *conceptualisation de base*, toutes les structures de données qui ne sont pas conceptuelles sont éliminées ou transformées. Elle est elle-même décomposée en :

- *détraduction du schéma* : le schéma de départ de ce processus est conforme au modèle du SGD utilisé, toutes ses structures de données spécifiques au SGD doivent être détectées et transformées en structures de données conceptuelles équivalentes;
- *désoptimisation du schéma* : le schéma a été restructuré et enrichi pour des raisons d'optimisation. Il faut détecter et éliminer ces structures.

Finalement nous disposons d'un schéma conceptuel qui peut encore être transformé, de manière à le rendre conforme à un standard méthodologique par exemple. Il s'agit d'un processus classique de *normalisation conceptuelle*.

### 3. Fragmentation de programmes

Nous décrivons brièvement une technique d'analyse et de transformation de programme utilisée dans la phase d'extraction des structures de données. Cette technique, appelée *fragmentation de programme* (*program slicing*), permet d'extraire d'un programme le fragment (idéalement) nécessaire et suffisant pour comprendre le comportement de ce programme à un point déterminé. Ce point est appelé *critère de fragmentation*. Un *fragment de programme* est constitué des parties du programme qui affectent les valeurs calculées par rapport au critère de fragmentation. Idéalement, un fragment est constitué de toutes les instructions qui affectent les valeurs calculées par rapport au critère de fragmentation et uniquement celles-là.

Le concept original de fragments de programmes a été introduit par Mark Weiser [Weiser 84]. Mark Weiser prétend qu'un fragment correspond à l'abstraction mentale d'un programmeur quand il débogue un programme. Différentes notions de fragmentation de programme et méthode de calcul ont été définies depuis, répondant chacune à des exigences particulières.

.....	1	ACCEPT N-CLI.	
FD CLIENT.	2	READ CLI KEY IS N-CLI.	
01 CLI.	3	MOVE 1 TO IND.	
02 N-CLI PIC 9(3).	4	MOVE 0 TO ORDER.	
02 NOM-CLI PIC X(10).	5	PERFORM UNTIL IND=10	
02 COM-CLI PIC 99 OCCURS 10.	6	ADD COM-CLI(IND) TO ORDER	
...	7	ADD 1 TO IND.	
01 ORDER PIC 9(3).	9	DISPLAY ORDER.	
...			
1		ACCEPT N-CLI.	(b)
2		READ CLI KEY IS N-CLI.	
3		MOVE 1 TO IND.	
4		MOVE 0 TO ORDER.	
5		PERFORM UNTIL IND=10	
6		ADD COM-CLI(IND) TO ORDER	
7		ADD 1 TO IND.	
8		DISPLAY NAME-CLI.	(c)
9		DISPLAY ORDER.	

(a)

Figure 2 : (a) un exemple de programme. (b) le fragment du programme par rapport à la ligne 9.  
(c) le fragment du programme par rapport à la ligne 8

La figure 2 (a) montre un exemple de programme qui acquiert un numéro (de client), lit dans le fichier l'enregistrement correspondant à ce client et affiche le nom et le montant total des commandes du client. La figure 2 (b) montre le fragment du programme par rapport à la ligne 9, seule la ligne qui affiche le nom du client ne fait pas partie de ce fragment. La figure 2 (c) est le

fragment du programme par rapport à la ligne 8, les instructions qui correspondent au calcul du montant total des commandes ne font pas partie de ce fragment car elles n'influencent pas la valeur du nom du client.

Les caractéristiques des différents langages de programmation comme les procédures, les contrôles de flux arbitraires (goto), les types de données composites et les pointeurs nécessitent des solutions spécifiques. Dans le cadre de la rétro-ingénierie de bases de données, nous sommes intéressés par les programmes COBOL avec des procédures, des types de données composées et des contrôles de flux arbitraires.

Nous nous sommes inspirés de la technique proposée par Susan Horwitz et al. [Horwitz et al. 90] pour le calcul de fragments inter-procéduraux. Ils ont redéfini le calcul d'un fragment en terme de parcours du *graphe de dépendance du système* (system dependence graph -- SDG). Le SDG est un graphe orienté dont les nœuds correspondent aux instructions et les arcs représentent les dépendances de données, les dépendances de contrôle et les appels de procédure. Un critère de fragmentation est identifié à un nœud du graphe. Un fragment est l'ensemble des nœuds du SDG desquels on peut atteindre le nœud représentant le critère de fragmentation.

La fragmentation de programme, telle que nous l'avons implémentée, calcule le fragment qui influence toutes les variables référencées dans le nœud qui est le critère de fragmentation. De plus, nous devons tenir compte de la manipulation des fichiers. Si, par exemple, un enregistrement est écrit dans un fichier et qu'une autre procédure (ou programme) lit un enregistrement de ce même fichier, comment savoir s'ils ont tous les deux lu le même enregistrement? Actuellement, nous ne sommes pas capables de faire ce lien, nous considérons qu'il s'agit de deux enregistrements différents. A l'avenir, il nous faudra aussi tenir compte des appels entre différents programmes et gérer les programmes qui accèdent aux mêmes fichiers. Notre fragmentation de programme ne travaille que sur un seul programme à la fois.

#### **4. L'outil DB-MAIN**

L'environnement d'ingénierie de bases de données DB-MAIN<sup>3</sup> est dédié à l'ingénierie des bases de données, ce qui englobe la rétro-ingénierie. En particulier son but est d'assister le développeur dans la conception, la rétro-ingénierie, la migration, la maintenance et l'évolution de bases de données. Dans cette section, nous citerons brièvement les principaux composants de cet atelier. Plus de détails peuvent être trouvés dans [Englebert et al. 95],[Hainaut et al. 93].

##### **4.1. Les fonctionnalités offertes par DB-MAIN**

Comme tout atelier de génie logiciel, DB-MAIN inclut les fonctions habituelles d'analyse et d'ingénierie, c'est-à-dire la création, la modification, l'affichage, la gestion, la validation et la transformation des spécifications ainsi que la génération de code, de rapports et des fonctions d'importation :

- Son élément central est un référentiel qui contient les définitions de tous les composants d'un projet.

---

<sup>3</sup>Pour Database Maintenance.

- L'interface graphique permet de manipuler ce référentiel et de demander l'exécution des opérations. L'interface peut être pilotée par un moteur méthodologique qui guide l'analyste dans le suivi d'une méthode d'ingénierie propre à son entreprise.
- DB-MAIN est basé sur une approche transformationnelle. Toutes les modifications appliquées à un schéma sont considérées comme des transformations et le processus de conception de bases de données est modélisé comme une séquence de transformations de schéma. Certaines transformations augmentent la sémantique d'un schéma (ajouter un type d'entités), d'autres la diminuent (supprimer un attribut). Les autres transformations qui préservent la sémantique d'un schéma, sont dites *réversibles*. Pour plus d'informations sur la notion de transformation, on consultera [Hainaut 81],[Hainaut et al. 94].
- Les extracteurs permettent d'extraire de façon automatique les structures de données déclarées dans un texte source. Actuellement, l'outil dispose de cinq extracteurs (SQL, COBOL, CODASYL, IMS, RPG).
- L'outil de recherche de patterns offre la recherche de patterns avec instanciation de variables. Les patterns sont exprimés dans un langage de définition de patterns (PDL) dérivé de la notation BNF.
- Le processeur de noms permet de transformer les noms d'un schéma ou de certains objets d'un schéma selon certaines règles de remplacement.
- Les assistants sont des outils de haut niveau dédiés à la résolution de problèmes spécifiques. DB-MAIN offre actuellement deux assistants, mais d'autres sont en développement. L'assistant de transformation permet d'appliquer une ou plusieurs transformations à des objets sélectionnés. L'assistant d'analyse permet d'analyser la conformité d'un schéma à un sous-modèle.
- DB-MAIN offre la possibilité d'enregistrer l'historique des modifications apportées à un schéma. Cet historique peut être rejoué sur le schéma de départ ou peut servir à reconstituer un historique inverse.
- DB-MAIN dispose d'un outil de fragmentation de programme qui permet d'extraire d'un texte source COBOL, pour un point  $p$ , toutes les instructions du programme qui peuvent affecter les variables référencées au point  $p$ .
- Un outil de calcul des dépendances des variables calcule les dépendances entre les variables d'un programme.
- Il apparaît rapidement que pour assurer la viabilité d'un outil, il est nécessaire de lui adjoindre un moyen de personnalisation tel un environnement de programmation permettant l'ajout dynamique de nouvelles fonctionnalités. Non seulement l'outil pourra communiquer avec d'autres, mais l'utilisateur pourra développer, et inclure dans l'atelier, des fonctions qui lui sont propres. C'est sur la base de cette constatation que nous avons défini le langage et l'environnement de programmation *Uoyager*<sup>2</sup>.

#### ***4.2. Les fonctionnalités de rétro-ingénierie offertes par DB-MAIN***

Les processus de rétro-ingénierie utilisent intensivement les fonctionnalités de DB-MAIN conçues pour la conception de bases de données. Parmi celles-ci, nous pouvons citer le modèle de représentation des données génériques, les différentes vues d'un même schéma, l'approche transformationnelle (via les transformations inverses), l'enregistrement de l'historique pour documenter et reconstituer un historique inverse.



Certains processeurs sont cependant spécifiques. Nous présenterons les outils de manipulation de textes, qui sont essentiellement (mais pas exclusivement) liés à la rétro-ingénierie.

Les textes sources fournissent une information importante pour la rétro-ingénierie, et font donc partie intégrante du projet. Plusieurs fonctions de présentation et d'analyse de ces textes ont été développées :

- Extraction automatique des structures de données à partir de textes SQL, COBOL, CODASYL, IMS, RPG. L'utilisateur peut également écrire ses propres extracteurs grâce au langage *Voyager2*.
- Affichage des textes sources.
- Le moteur de recherche permet de repérer, dans des fichiers sources ou les descriptions d'un schéma, des patterns définis dans un langage de définition (*Pattern Definition Language* -- PDL). Le PDL est proche de la notation BNF et dispose de variables qui sont instanciables avant ou pendant la recherche d'un pattern. Ces patterns peuvent simplement être utilisés pour effectuer une recherche. Ils peuvent aussi être couplés à des procédures *Voyager2*, qui sont exécutées lorsqu'un pattern a été trouvé et qui utilisent comme paramètres d'entrée les valeurs des variables du pattern.
- Un outil permet de construire et consulter le graphe de dépendance des variables d'un texte source. Le graphe de dépendance des variables est un graphe où chaque variable est représentée par un nœud et où les arcs représentent une relation entre deux variables (assignation, comparaison, etc.). S'il existe un chemin entre les variables *A* et *B*, c'est que *A* dépend (directement ou indirectement) de *B*, ou inversement. Dans DB-MAIN, le graphe n'est pas orienté et l'utilisateur donne une liste de patterns qui définissent les relations entre les variables.
- Enrichissement du schéma à partir de structures de données découvertes dans les textes sources. En sélectionnant les définitions de variables dans le texte source, DB-MAIN peut créer automatiquement les attributs correspondants dans le schéma.
- Outil de navigation des schémas vers les textes sources et inversement (parcours des correspondances).
- Outil de fragmentation de programmes qui permet d'extraire d'un texte source COBOL, pour un point *p*, toutes les instructions du programme qui peuvent affecter les variables référencées au point *p*.

## 5. Une étude de cas

Cette section est consacrée à une étude de cas complète de rétro-ingénierie d'une application COBOL, dont le texte source se trouve en annexe. Nous allons appliquer à ce programme notre méthode générique de rétro-ingénierie.

La partie la plus ardue du travail est l'extraction des structures de données car les seules contraintes que COBOL peut vérifier sont les identifiants. L'extraction automatique des structures de données ne nous donne que peu d'informations : la liste des fichiers et des enregistrements avec leurs attributs et leurs identifiants. Pour compléter l'extraction des structures de données, il faut analyser le texte source. Cette tâche, bien que facilitée par l'existence du moteur de recherche, du graphe de dépendance des variables et de l'outil de fragmentation de programme, est longue et fastidieuse.

Lorsque l'extraction est terminée, la conceptualisation du schéma ne pose plus de problèmes grâce aux outils de transformations offerts par DB-MAIN.

### 5.1. Extraction des structures de données

L'extraction des structures de données dans le cas d'une application COBOL repose principalement sur l'analyse du texte source pour détecter toutes les structures de données.

L'extracteur automatique donne la structure des fichiers déclarés dans la FILE-CONTROL et la FILE SECTION du programme, c'est-à-dire la liste des fichiers, des enregistrements, des attributs, des identifiants et des clés d'accès. Le résultat de cette extraction automatique donne ce que nous appelons le schéma logique brut. Ce schéma doit être complété pour obtenir le schéma logique qui contient toutes les contraintes d'intégrités de la base de données. Les contraintes à retrouver lors de l'analyse du texte source sont principalement des champs dont la structure n'est pas complète, les contraintes référentielles, les identifiants des champs multivalués, la cardinalité exacte des champs multivalués.

#### 5.1.1. Extraction

La première étape de l'extraction des structures de données consiste en l'extraction de la FILE-CONTROL et de la FILE SECTION du texte source COBOL. Cette extraction peut être faite de façon totalement automatique au moyen de l'extracteur COBOL de DB-MAIN.

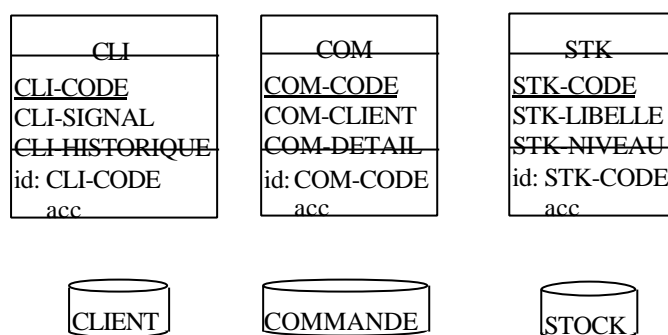


Figure 3 : Le schéma logique brut

La figure 3 représente le résultat obtenu par l'extracteur COBOL de DB-MAIN. Les fichiers sont représentés par des cylindres. Les enregistrements sont représentés par des rectangles divisés en trois. La partie supérieure contient le nom de l'enregistrement, celle du milieu les champs de l'enregistrement (avec l'identifiant souligné). Celle du bas représente les contraintes, *id* signifie que l'attribut est identifiant et *acc* qu'il existe un index sur cet attribut.

#### 5.1.2. Affinement des champs

Les structures obtenues lors de l'extraction automatique sont incomplètes. Certains des champs ont une longueur "trop" grande (CLI-SIGNAL, CLI-HISTORIQUE, COM-DETAIL, STK-LIBELLE). Grâce au graphe de dépendance des variables, on peut détecter que certaines de ces variables dépendent d'autres variables dont la structure est plus fine.

Si l'on trouve, dans le graphe de dépendance des variables (figure 4), une variable qui est en relation directe ou indirecte avec une autre de ces variables et a une structure plus fine, on aura trouvé une décomposition de cette variable.

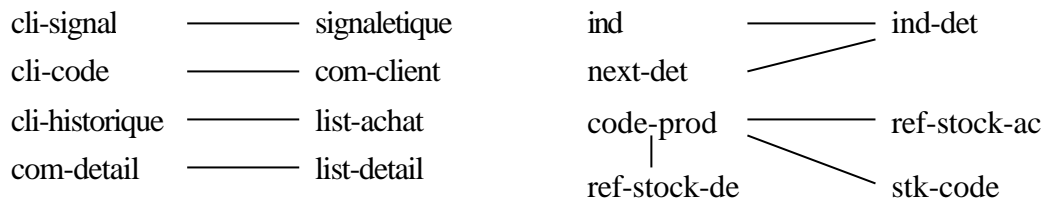


Figure 4 : Le graphe de dépendance des variables

Par exemple, CLI-SIGNAL peut être décomposée selon la même structure que SIGNALETIQUE. On peut ainsi affiner la structure de CLI-SIGNAL comme celle de SIGNALETIQUE. On peut recommencer la même opération avec les autres variables et on découvre que :

- CLI-HISTORIQUE peut être affinée comme LIST-ACHAT;
- COM-DETAIL peut être affinée comme LIST-DETAIL;
- STK-LIBELLE ne doit pas être affiné, car STK-LIBELLE n'est en relation avec aucune autre variable.

#### 5.1.3. Détection des contraintes référentielles

La recherche de contraintes référentielles est la recherche des liens qui existent entre deux fichiers d'une application. Pour cela, il existe plusieurs méthodes complémentaires :

1. le nom d'un champ inclut le nom du fichier ou enregistrement référencé;
2. un champ a la même longueur que l'identifiant du fichier référencé;
3. il existe un index sur ce champ;
4. il existe, dans le graphe de dépendance des variables, un chemin entre un champ et un identifiant du fichier référencé;
5. le contenu d'un enregistrement dépend du contenu d'un autre fichier.

Nous allons décrire comment retrouver les enregistrements qui respectent les points 4 et 5 en utilisant deux outils offerts par DB-MAIN.

Pour trouver la dépendance entre un identifiant et un autre champ, il suffit, lorsque le graphe de dépendance des variables a été construit, de déterminer si les identifiants sont liés à un champ d'un autre fichier. Si c'est le cas, on peut faire l'hypothèse qu'il existe une clé étrangère entre le champ trouvé et l'identifiant sélectionné. Pour s'en assurer, nous allons calculer et analyser le fragment de programme par rapport à l'instruction qui fait le lien entre les deux champs.

Pour détecter s'il y a un lien entre un enregistrement et un autre fichier, il faut calculer le fragment de programme par rapport aux instructions d'écriture et de lecture de l'enregistrement. Si ces fragments contiennent une instruction de lecture d'un autre fichier, il y a une contrainte référentielle entre les deux fichiers.

```

181  NOUV-COM.
186  MOVE 1 TO END-FILE.
187  PERFORM READ-CODE-CLI UNTIL END-FILE = 0.
190  MOVE CLI-CODE TO COM-CLIENT.

203  READ-CODE-CLI.
205  ACCEPT CLI-CODE.
206  MOVE 0 TO END-FILE.
207  READ CLIENT INVALID KEY
209  MOVE 1 TO END-FILE
210  END-READ.

```

Figure 5 : Le fragment de programme par rapport à la ligne 190

Dans notre exemple, CLI-CODE et COM-CLIENT sont liés dans le graphe de dépendance des variables. COM-CLIENT a la même longueur que CLI-CODE et contient le nom de l'enregistrement de CLI-CODE. Tous ces éléments en font un bon candidat pour être une clé étrangère. On trouve que la ligne 190 fait le lien entre les deux variables. On calcule le fragment du programme par rapport à cette ligne, ce qui donne le fragment de la figure 5.

On peut remarquer que l'on exécute la procédure READ-CODE-CLI (ligne 187) jusqu'à ce que END-FILE soit égale à 0. END-FILE ne prend cette valeur que si l'utilisateur donne à CLI-CODE (ligne 205) une valeur qui existe dans CLIENT (ligne 206, 207, 209). Donc COM-CLIENT a obligatoirement une valeur qui existe dans CLI-CODE. Cela confirme qu'il existe une contrainte référentielle de COM-CLIENT vers CLI-CODE.

181	NOUV-COM.	226	IF EXIST-PROD = 0
184	ACCEPT COM-CODE.	228	ELSE
186	MOVE 1 TO END-FILE.	229	PERFORM UPDATE-COM-DETAIL.
187	PERFORM READ-CODE-CLI UNTIL END-FILE = 0.	231	UPDATE-COM-DETAIL.
190	MOVE CLI-CODE TO COM-CLIENT.	232	MOVE 1 TO NEXT-DET.
193	SET IND-DET TO 1.	234	ACCEPT Q-COM(IND-DET).
194	MOVE 1 TO END-FILE.	235	PERFORM UNTIL
195	PERFORM READ-DETAIL UNTIL END-FILE=0 OR IND-DET=21.	236	REF-STOCK-DE(NEXT-DET) = CODE-PROD
196	MOVE LIST-DETAIL TO COM-DETAIL.	237	OR IND-DET = NEXT-DET
198	WRITE COM INVALID KEY DISPLAY "ERREUR".	238	ADD 1 TO NEXT-DET
203	READ-CODE-CLI.	239	END-PERFORM.
205	ACCEPT CLI-CODE.	240	IF IND-DET = NEXT-DET
206	MOVE 0 TO END-FILE.	241	MOVE CODE-PROD TO REF-STOCK-DE(IND-DET)
207	READ CLIENT INVALID KEY	243	SET IND-DET UP BY 1.
209	MOVE 1 TO END-FILE.	247	UPDATE-CLI-HISTO.
212	READ-DETAIL.	248	SET IND TO 1.
214	ACCEPT CODE-PROD.	249	PERFORM UNTIL
215	IF CODE-PROD = "0"	250	REF-STOCK-AC(IND) = CODE-PROD
216	MOVE 0 TO END-FILE	251	OR REF-STOCK-AC(IND) = 0 OR IND = 101
217	MOVE 0 TO REF-STOCK-DE(IND-DET)	252	SET IND UP BY 1
218	ELSE	253	END-PERFORM.
219	PERFORM READ-CODE-PROD.	257	IF REF-STOCK-AC(IND) = CODE-PROD
221	READ-CODE-PROD.	258	ADD Q-COM(IND-DET) TO TOT(IND).
222	MOVE 1 TO EXIST-PROD.	259	ELSE
223	MOVE CODE-PROD TO STK-CODE.	261	MOVE Q-COM(IND-DET) TO TOT(IND).
224	READ STOCK INVALID KEY		
225	MOVE 0 TO EXIST-PROD.		

Figure 6 : Le fragment par rapport à la ligne 198

181	NOUV-COM.	212	READ-DETAIL.
193	SET IND-DET TO 1.	214	ACCEPT CODE-PROD.
194	MOVE 1 TO END-FILE.	215	IF CODE-PROD = "0"
195	PERFORM READ-DETAIL UNTIL END-FILE=0 OR IND-DET=21.	218	ELSE
		219	PERFORM READ-CODE-PROD.
		221	READ-CODE-PROD.
		223	MOVE CODE-PROD TO STK-CODE.
		224	READ STOCK INVALID KEY

Figure 7 : Le fragment par rapport à la ligne 224

Continuons à analyser l'enregistrement COM. Si l'on calcule le fragment de programme par rapport à l'instruction d'écriture de l'enregistrement (ligne 198 – figure 6), on retrouve bien sûr que la valeur de COM dépend de CLIENT mais aussi de STOCK (ligne 224). En observant le fragment, on peut constater que la valeur de STOCK (ou d'un de ses champs) n'influence pas directement la valeur de COM. Pour connaître le lien qui existe entre COM et STOCK, on commence par calculer le fragment de programme par rapport à la ligne 224 (figure 7) pour savoir quelles sont les variables qui influencent la ligne 224. On remarque que l'utilisateur donne une valeur à CODE-PROD et que cette valeur sert de clé d'accès à STOCK. Grâce au graphe de dépendance et à l'analyse du fragment (figure 6), on constate que CODE-PROD est assigné à

REF-STOCK-DE (ligne 241) qui appartient à LIST-DETAIL qui lui même est assigné à COM-DETAIL (ligne 196). De plus, on ne peut accéder à la ligne 241 que s'il existe un élément de STOCK qui a CLI-CODE égale à CODE-PROD, ce qui implique que tous les éléments de REF-STOCK-DE appartiennent à STOCK. Nous venons de trouver une contrainte référentielle.

181	NOUV-COM.	225	MOVE 0 TO EXIST-PROD.
186	MOVE 1 TO END-FILE.	226	IF EXIST-PROD = 0
187	PERFORM READ-CODE-CLI UNTIL END-FILE = 0.	228	ELSE
188	MOVE CLI-SIGNAL TO SIGNALETIQUE.	229	PERFORM UPDATE-COM-DETAIL.
191	MOVE CLI-HISTORIQUE TO LIST-ACHAT.	231	UPDATE-COM-DETAIL.
193	SET IND-DET TO 1.	232	MOVE 1 TO NEXT-DET.
194	MOVE 1 TO END-FILE.	234	ACCEPT Q-COM(IND-DET).
195	PERFORM READ-DETAIL UNTIL END-FILE=0 OR IND-DET=21.	235	PERFORM UNTIL
200	MOVE LIST-ACHAT TO CLI-HISTORIQUE.	236	REF-STOCK-DE(NEXT-DET) = CODE-PROD
201	REWRITE CLI INVALID KEY DISPLAY "ERREUR CLI".	237	OR IND-DET = NEXT-DET
203	READ-CODE-CLI.	238	ADD 1 TO NEXT-DET
205	ACCEPT CLI-CODE.	239	END-PERFORM.
206	MOVE 0 TO END-FILE.	240	IF IND-DET = NEXT-DET
207	READ CLIENT INVALID KEY	242	PERFORM UPDATE-CLI-HISTO
209	MOVE 1 TO END-FILE	243	SET IND-DET UP BY 1
210	END-READ.	247	UPDATE-CLI-HISTO.
212	READ-DETAIL.	248	SET IND TO 1.
214	ACCEPT CODE-PROD.	249	PERFORM UNTIL
215	IF CODE-PROD = "0"	250	REF-STOCK-AC(IND) = CODE-PROD
216	MOVE 0 TO END-FILE	251	OR REF-STOCK-AC(IND) = 0 OR IND = 101
217	MOVE 0 TO REF-STOCK-DE(IND-DET)	252	SET IND UP BY 1
218	ELSE	253	END-PERFORM.
219	PERFORM READ-CODE-PROD.	257	IF REF-STOCK-AC(IND) = CODE-PROD
221	READ-CODE-PROD.	258	ADD Q-COM(IND-DET) TO TOT(IND)
222	MOVE 1 TO EXIST-PROD.	259	ELSE
223	MOVE CODE-PROD TO STK-CODE.	260	MOVE CODE-PROD TO REF-STOCK-AC(IND)
224	READ STOCK INVALID KEY	261	MOVE Q-COM(IND-DET) TO TOT(IND).

Figure 8 : Le fragment par rapport à la ligne 201

L'enregistrement CLI est écrit à deux endroits (lignes 134 et 201). Si l'on calcule les fragments de programme par rapport à ces deux lignes, l'on constate que la ligne 134 n'est influencée par aucun fichier, par contre la ligne 201 (figure 8) est influencée par le fichier STOCK (ligne 224) tout comme l'était COM. On peut reprendre le même raisonnement que dans le paragraphe précédent. CODE-PROD est assigné à REF-STOCK-AC (ligne 261) et cette instruction ne peut être atteinte que si CODE-PROD appartient à STOCK. Il existe donc une contrainte référentielle de REF-STOCK-AC vers COM-CODE.

#### 5.1.4. Recherche des identifiants des champs multivalués

Pour chaque champ multivalué décomposable en sous-champs, il faut déterminer si l'un de ces sous-champs est un identifiant local au champ multivalué. Pour connaître si un champ à un identifiant local, on va calculer le fragment du programme par rapport à la dernière instruction qui référence le champ multivalué avant son écriture.

Dans notre exemple, il existe deux champs multivalués ACHAT et DETAILS.

<pre> 181 NOUV-COM. 193 SET IND-DET TO 1. 194 MOVE 1 TO END-FILE. 195 PERFORM READ-DETAIL UNTIL END-FILE=0 OR IND-DET=21. 196 MOVE LIST-DETAIL TO COM-DETAIL. 212 READ-DETAIL. 214 ACCEPT CODE-PROD. 215 IF CODE-PROD = "0" 216     MOVE 0 TO END-FILE 217     MOVE 0 TO REF-STOCK-DE(IND-DET) 218 ELSE 219     PERFORM READ-CODE-PROD. 221 READ-CODE-PROD. 222 MOVE 1 TO EXIST-PROD. 223 MOVE CODE-PROD TO STK-CODE. 224 READ STOCK INVALID KEY 225     MOVE 0 TO EXIST-PROD. 226 IF EXIST-PROD = 0 228 ELSE 229     PERFORM UPDATE-COM-DETAIL. </pre>	<pre> 231 UPDATE-COM-DETAIL. 232 MOVE 1 TO NEXT-DET. 234 ACCEPT Q-COM(IND-DET). 235 PERFORM UNTIL 236     REF-STOCK-DE(NEXT-DET) = CODE-PROD 237     OR IND-DET = NEXT-DET 238     ADD 1 TO NEXT-DET 239 END-PERFORM. 240 IF IND-DET = NEXT-DET 241     MOVE CODE-PROD TO REF-STOCK-DE(IND-DET) 242     PERFORM UPDATE-CLI-HISTO 243     SET IND-DET UP BY 1. 247 UPDATE-CLI-HISTO. 248 SET IND TO 1. 249 PERFORM UNTIL 250     REF-STOCK-AC(IND) = CODE-PROD 251     OR REF-STOCK-AC(IND) = 0 OR IND = 101 252     SET IND UP BY 1 253 END-PERFORM. 257 IF REF-STOCK-AC(IND) = CODE-PROD 258     ADD Q-COM(IND-DET) TO TOT(IND) 259 ELSE 260     MOVE CODE-PROD TO REF-STOCK-AC(IND) 261     MOVE Q-COM(IND-DET) TO TOT(IND). </pre>
---	---

Figure 9 : Le fragment par rapport à la ligne 196

Nous allons commencer par analyser DETAILS. On calcule le fragment de programme par rapport à la ligne 196 (figure 9, dernière ligne avant l'écriture de l'enregistrement qui contient DETAILS et qui le référence). DETAILS a deux champs REF-STOCK-DE et Q-COM. Il n'y a aucun contrôle sur la valeur de Q-COM (ligne 234). La ligne 241 est la seule qui modifie REF-STOCK-DE. Elle n'est exécutée que s'il n'y a pas encore d'élément REF-STOCK-DE qui a la valeur CODE-PROD dans les IND-DET premiers éléments de DETAILS (lignes 235-239) et que si les autres éléments n'ont pas encore été assignés. On peut en conclure que REF-STOCK-DE est l'identifiant de DETAILS.

<pre> 181 NOUV-COM. 186 MOVE 1 TO END-FILE. 187 PERFORM READ-CODE-CLI UNTIL END-FILE = 0. 191 MOVE CLI-HISTORIQUE TO LIST-ACHAT. 193 SET IND-DET TO 1. 194 MOVE 1 TO END-FILE. 195 PERFORM READ-DETAIL UNTIL END-FILE = 0 OR IND-DET=21. 200 MOVE LIST-ACHAT TO CLI-HISTORIQUE. 203 READ-CODE-CLI. 205 ACCEPT CLI-CODE. 206 MOVE 0 TO END-FILE. 207 READ CLIENT INVALID KEY 209     MOVE 1 TO END-FILE 210 END-READ. 212 READ-DETAIL. 214 ACCEPT CODE-PROD. 215 IF CODE-PROD = "0" 216     MOVE 0 TO END-FILE 217     MOVE 0 TO REF-STOCK-DE(IND-DET) 218 ELSE 219     PERFORM READ-CODE-PROD. 221 READ-CODE-PROD. 222 MOVE 1 TO EXIST-PROD. 223 MOVE CODE-PROD TO STK-CODE. </pre>	<pre> 224 READ STOCK INVALID KEY 225 MOVE 0 TO EXIST-PROD. 226 IF EXIST-PROD = 0 228 ELSE 229     PERFORM UPDATE-COM-DETAIL. 231 UPDATE-COM-DETAIL. 232 MOVE 1 TO NEXT-DET. 234 ACCEPT Q-COM(IND-DET). 235 PERFORM UNTIL 236     REF-STOCK-DE(NEXT-DET) = CODE-PROD 237     OR IND-DET = NEXT-DET 238     ADD 1 TO NEXT-DET 239 END-PERFORM. 240 IF IND-DET = NEXT-DET 242     PERFORM UPDATE-CLI-HISTO 243     SET IND-DET UP BY 1. 247 UPDATE-CLI-HISTO. 248 SET IND TO 1. 249 PERFORM UNTIL 250     REF-STOCK-AC(IND) = CODE-PROD 251     OR REF-STOCK-AC(IND) = 0 OR IND = 101 252     SET IND UP BY 1 253 END-PERFORM. 257 IF REF-STOCK-AC(IND) = CODE-PROD 258     ADD Q-COM(IND-DET) TO TOT(IND) 259 ELSE 260     MOVE CODE-PROD TO REF-STOCK-AC(IND) 261     MOVE Q-COM(IND-DET) TO TOT(IND). </pre>
---	--

Figure 10 : Le fragment par rapport à la ligne 200

Pour trouver l'identifiant de ACHAT, on calcule le fragment de programme par rapport à la ligne 200 (figure 9), dernière ligne avant l'écriture de l'enregistrement (CLIENT) qui contient ACHAT.

ACHAT a deux champs REF-STOCK-AC et TOT. TOT est une valeur calculée à partir de la valeur du dernier Q-COM (ligne 258 et 262), ces instructions peuvent être utilisées pour déterminer les règles de calcul de TOT et ajouter une contrainte d'intégrité au schéma. La ligne 261 est la seule qui modifie REF-STOCK-AC et elle n'est exécutée que s'il n'y a pas d'autres éléments de ACHAT qui ont la valeur CODE-PROD (ligne 249-253). REF-STOCK-AC est un identifiant de ACHAT. De plus, toutes les valeurs de REF-STOCK-AC font partie de REF-STOCK-DE pour un client donné.

#### 5.1.5. Calcul de la cardinalité exacte des champs multivalués

Les champs multivalués proviennent d'une clause OCCURS lors de la déclaration de la variable. En COBOL, comme dans la plupart des langages de programmation, les tableaux ont une longueur fixe et il n'y a pas de possibilité pour indiquer si un élément est facultatif. Si certains éléments du tableau sont facultatifs, c'est au programmeur de les gérer de façon explicite dans le code procédural.

Dans notre exemple, nous avons deux champs multivalués DETAILS et ACHAT.

Comme nous l'avons vu dans la section précédente lors du garnissage de DETAILS seuls les premiers éléments ont une valeur significative, si l'utilisateur décide de ne pas garnir tous les éléments, alors le premier élément libre est garni avec la valeur "0" pour REF-STOCK-DE (figure 9 – lignes 215, 217). Une autre façon d'arriver à la même conclusion est d'analyser le paragraphe DISPLAY-DETAIL qui parcourt les éléments de DETAILS jusqu'à ce que REF-STOCK-DE soit égal à "0". Cela permet de déterminer que DETAILS a une cardinalité minimum de 0.

Dans la section précédente, on a également remarqué que l'on parcourait ACHAT jusqu'à rencontrer le premier élément égal à "0" (figure 10, ligne 248-253). Donc ACHAT a une cardinalité minimum de 0.

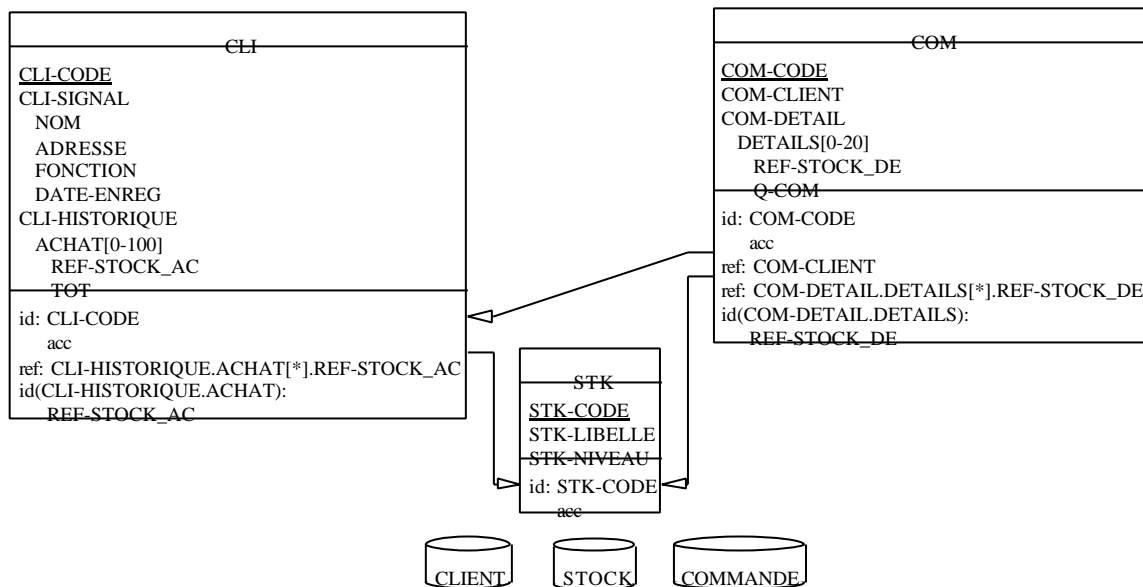


Figure 11 : Le schéma logique

On obtient ainsi le schéma logique de la figure 11. Ce schéma est conforme au fichier COBOL avec en plus toutes les contraintes d'intégrités qui ont été trouvées lors de l'analyse du texte source ou dans d'autres sources d'informations.

## 5.2. Conceptualisation du schéma

La conceptualisation du schéma peut entièrement être faite grâce aux transformations présentes dans DB-MAIN.

### 5.2.1. Préparation du schéma

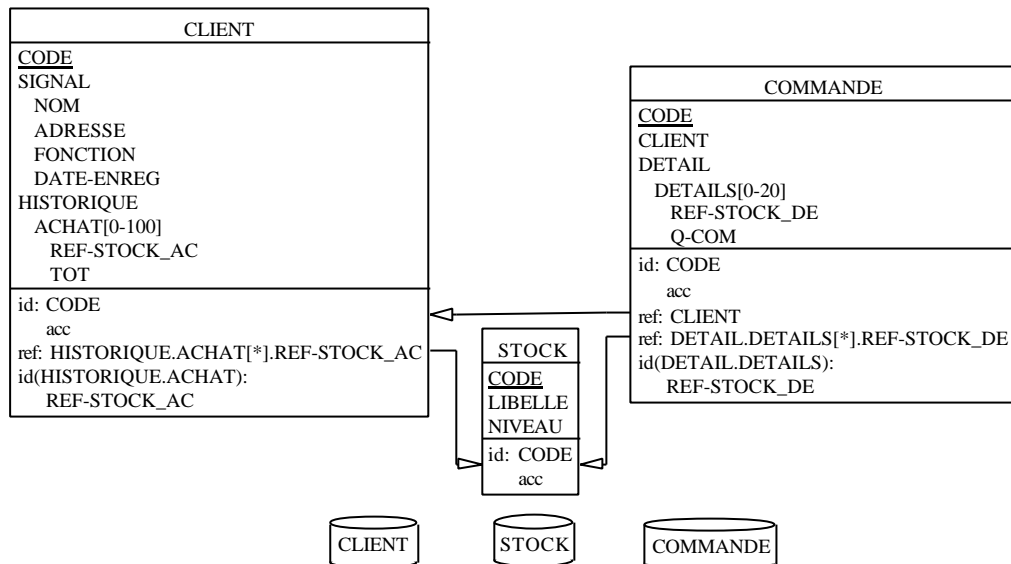


Figure 12 : Le schéma préparé

Avant de transformer notre schéma logique en un schéma conceptuel, on va modifier certains noms d'attributs et de types d'entités de manière à les rendre plus significatifs (figure 12) :

- les fichiers ont des noms plus significatifs que ceux des enregistrements, on remplace les noms des types d'entités par le nom du fichier correspondant;
- les attributs sont préfixés par le nom du type d'entités; on élimine ces préfixes grâce à la fonction de dépréfixage de DB-MAIN.

### 5.2.2. Conceptualisation de base

On peut éliminer des constructions physiques qui n'ont plus de sens au niveau conceptuel :

- suppression des fichiers;
- suppression des index;
- les champs décomposables CLI-HISTORIQUE et COM-DETAIL n'ont qu'un seul composant et peuvent donc être désagrégés sans perte de sémantique.



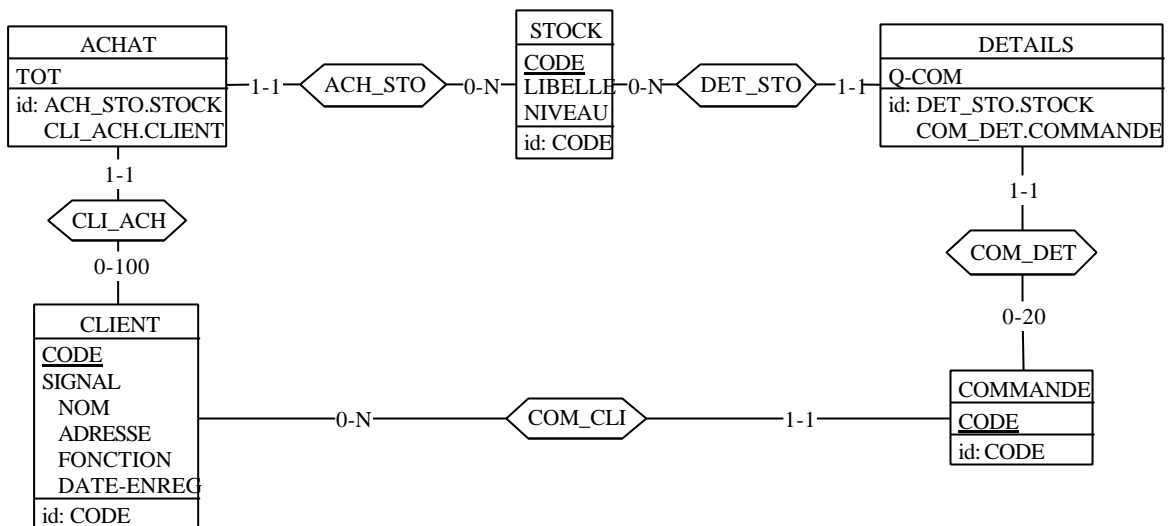


Figure 13 : Le schéma conceptuel brut

Les attributs DETAILS et ACHAT ont une structure complexe :

- ils sont décomposables;
- ils sont multivalués;
- ils ont un identifiant local;
- ils ont une clé étrangère.

Tous ces éléments font penser qu'ils représentent très certainement l'implémentation COBOL de types d'entités dépendants. On peut les transformer en types d'entités au moyen d'une transformation de l'atelier.

Les seuls éléments qu'il reste à détraduire sont les clés étrangères à transformer en types d'associations.

La figure 13 représente le schéma conceptuel brut.

### 5.2.3. Normalisation conceptuelle

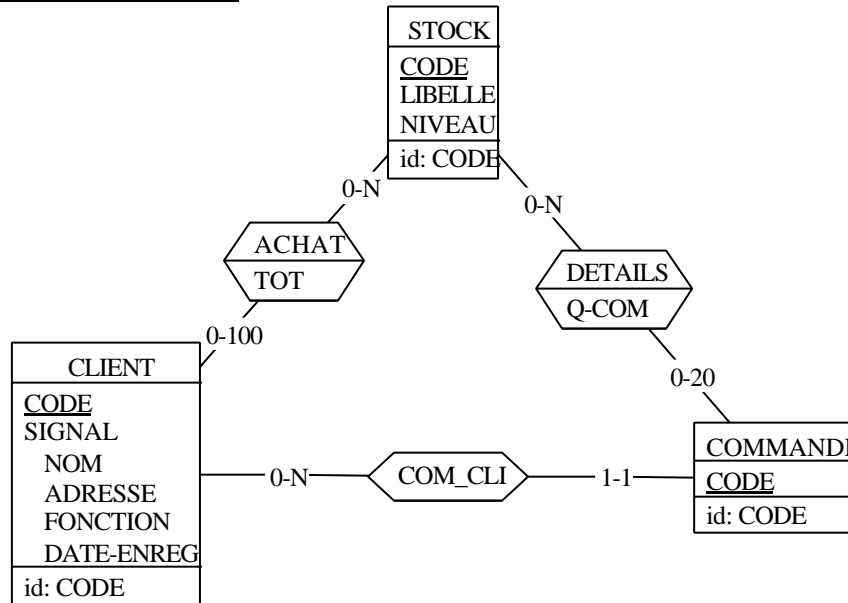


Figure 14 : Le schéma conceptuel

Pour affiner le schéma conceptuel, on peut encore transformer les types d'entités ACHAT et DETAILS en types d'associations.

On obtient finalement le schéma conceptuel de la figure 14.

## 6. Conclusion

Comme le montre l'étude de cas, la rétro-ingénierie de bases de données, même pour un programme en apparence simple, est un processus long et compliqué. Pour être mené à bien, pour des projets de grandeur réelle, ce processus réclame des outils puissants, flexibles et adaptés car les sources d'informations sont volumineuses et diversifiées.

L'outil DB-MAIN présenté dans cet article essaye de couvrir au maximum le processus de rétro-ingénierie de bases de données. Bien qu'encore en développement, DB-MAIN offre déjà les outils nécessaires à la rétro-ingénierie d'applications de complexité réaliste.

Dans sa version actuelle DB-MAIN offre :

- des projets multi-schémas;
- six types de présentations de schémas;
- plus de 25 transformations, pour la normalisation, l'optimisation, la restructuration, la rétro-ingénierie, etc.;
- des assistants de transformation et d'analyse;
- la génération de code exécutable et de rapports;
- des extracteurs pour SQL, COBOL, CODASYL, IMS et RPG;
- des outils de manipulation de texte sources (recherche de patterns, création automatique d'attributs à partir du texte source, calcul du graphe de dépendance des variables, fragmentation de programme);
- un langage de programmation (*Voyager2*);
- la journalisation des actions de l'utilisateur et la possibilité de rejouer un journal.

DB-MAIN est développé en C++ sous MS-Windows. Il existe une version de démonstration (complète mais ne permettant de manipuler que des schémas limités en taille) disponible pour

évaluation. Toutes informations complémentaires concernant DB-MAIN peuvent être obtenues en contactant db-main@info.fundp.ac.be.

## Références

- [Andersson 94] M. Andersson. Extracting an entity relationship schema from a relational database through reverse engineering. In *Proc. of the 13th Int. Conf. on ERA*, Manchester, 1994. Springer-Verlag.
- [Clarinval 91] A. Clarinval. Comprendre et Connaître le COBOL 85. 1991, Presses Universitaires de Namur.
- [Englebert et al. 95] V. Englebert, J. Henrard, Hick J-M, D. Roland, and J-L Hainaut. DB-MAIN: Un atelier d'ingénierie de bases de données. In *Onzième Journées Bases de Données Avancées*, Nancy – France, 1995.
- [Hainaut 81] J-L Hainaut. Theoretical and practical tools for database design. In *Proc. Int. VLDB Conf. ACM/IEEE*, 1981.
- [Hainaut et al. 93] J-L Hainaut, M. Chandelon, C. Tonneau, and M. Joris. Contribution to a theory of database reverse engineering. In *Proc of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993. IEEE CSP.
- [Hainaut et al. 94] J-L Hainaut, V. Englebert, J. Henrard, Hick J-M, and D. Roland. Evolution of database applications : The DB-MAIN approach. In *Proc. of the 13th Int. Conf. on ERA*, Manchester, 1994. Springer-Verlag.
- [Hainaut et al. 95] J-L. Hainaut, V. Englebert, J. Henrard, Hick J-M., and D. Roland. Requirements for information system reverse engineering support. In *Proc. of the IEEE Working Conference on Reverse Engineering*, IEEE Computer Society Press, Toronto, July 1995.
- [Horwitz et al. 90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [Petit et al. 94] J-M. Petit, J. Kouloumdjian, J-F. Bouliant, and F. Toumani. Using queries to improve database reverse engineering. In *Proc. of the 13th Int. Conf. on ERA*, Manchester, 1994. Springer-Verlag.
- [Tip 94] F. Tip. A survey of program slicing techniques. Technical report, CWI, 94.
- [Weiser 84] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, SE-10(4):352–357, July 1984.

## Annexe : Le programme

```
1  IDENTIFICATION DIVISION.
2  PROGRAM-ID. CLIENT-COMMANDE.
3
4  ENVIRONMENT DIVISION.
5  INPUT-OUTPUT SECTION.
6  FILE-CONTROL.
7  SELECT CLIENT ASSIGN TO "CLIENT.DAT"
8  ORGANIZATION IS INDEXED
9  ACCESS MODE IS DYNAMIC
10 RECORD KEY IS CLI-CODE,11
12 SELECT COMMANDE ASSIGN TO "COMMANDE.DAT"
13 ORGANIZATION IS INDEXED
14 ACCESS MODE IS DYNAMIC
15 RECORD KEY IS COM-CODE.
16
17 SELECT STOCK ASSIGN TO "STOCK.DAT"
18 ORGANIZATION IS INDEXED
19 ACCESS MODE IS DYNAMIC
20 RECORD KEY IS STK-CODE.
21
22 DATA DIVISION.
23 FILE SECTION.
24 FD CLIENT.
25 01 CLI.
26 02 CLI-CODE PIC X(12).
27 02 CLI-SIGNAL PIC X(80).
28 02 CLI-HISTORIQUE PIC X(1000).
29
30 FD COMMANDE.
31 01 COM.
32 02 COM-CODE PIC 9(10).
33 02 COM-CLIENT PIC X(12).
34 02 COM-DETAIL PIC X(200).
35
36 FD STOCK.
37 01 STK.
```

```

38      02 STK-CODE PIC 9(5).
39      02 STK-LIBELLE PIC X(100).
40      02 STK-NIVEAU PIC 9(5).
41
42
43 WORKING-STORAGE SECTION.
44 01 SIGNALETIQUE.
45     02 NOM PIC X(20).
46     02 ADRESSE PIC X(40).
47     02 FONCTION PIC X(10).
48     02 DATE-ENREG PIC X(10).
49
50 01 LIST-ACHAT.
51     02 ACHAT OCCURS 100 TIMES INDEXED BY IND.
52     03 REF-STOCK-AC PIC 9(5).
53     03 TOT PIC 9(5).
54
55 01 LIST-DETAIL.
56     02 DETAILS OCCURS 20 TIMES INDEXED BY IND-DET.
57     03 REF-STOCK-DE PIC 9(5).
58     03 Q-COM PIC 9(5).
59
60 01 CHOIX PIC X.
61
62 01 END-FILE PIC 9.
63
64 01 END-DETAIL PIC 9.
65
66 01 EXIST-PROD PIC 9.
67
68 01 CODE-PROD PIC 9(5).
69
70 01 QTE PIC 9(5) COMP.
71
72 01 NEXT-DET PIC 99.
73
74 PROCEDURE DIVISION.
75 MAIN.
76     PERFORM INIT.
77     PERFORM TRAITEMENT UNTIL CHOIX = 0.
78     PERFORM CLOTURE.
79     STOP RUN.
80
81 INIT.
82     OPEN I-O CLIENT.
83     OPEN I-O COMMANDE.
84     OPEN I-O STOCK.
85
86 TRAITEMENT.
87     DISPLAY "1 NOUVEAU CLIENT".
88     DISPLAY "2 NOUVEAU STOCK".
89     DISPLAY "3 NOUVELLE COMMANDE".
90     DISPLAY "4 LISTE DES CLIENTS".
91     DISPLAY "5 LISTE DU STOCK".
92     DISPLAY "6 LISTE DES COMMANDES".
93     DISPLAY "0 FIN".
94     ACCEPT CHOIX.
95
96     IF CHOIX = 1
97         PERFORM NOUV-CLI.
98
99     IF CHOIX = 2
100        PERFORM NOUV-STK.
101
102     IF CHOIX = 3
103        PERFORM NOUV-COM.
104
105     IF CHOIX = 4
106        PERFORM LIST-CLI.
107
108     IF CHOIX = 5
109        PERFORM LIST-STK.
110
111     IF CHOIX = 6
112        PERFORM LIST-DET.
113
114 CLOTURE.
115     CLOSE CLIENT.
116     CLOSE COMMANDE.
117     CLOSE STOCK.
118
119 NOUV-CLI.
120     DISPLAY "NOUVEAU CLIENT : ".
121     DISPLAY "CODE DU CLIENT ?" WITH NO ADVANCING.
122     ACCEPT CLI-CODE.
123
124     DISPLAY "NOM DU CLIENT : " WITH NO ADVANCING.
125     ACCEPT NOM.
126     DISPLAY "ADRESSE DU CLNT:" WITH NO ADVANCING.
127     ACCEPT ADRESSE.
128     DISPLAY "FONCTION DU CLNT:" WITH NO ADVANCING.
129     ACCEPT FONCTION.
130     DISPLAY "DATE : " WITH NO ADVANCING.
131     ACCEPT DATE-ENREG.
132     MOVE SIGNALETIQUE TO CLI-SIGNAL.
133     DISPLAY CLI-SIGNAL.
134
135     PERFORM INIT-HISTO.
136
137     WRITE CLI INVALID KEY DISPLAY "ERREUR".
138
139 LIST-CLI.
140     DISPLAY "LISTE DES CLIENTS".
141     CLOSE CLIENT.
142     OPEN I-O CLIENT.
143     MOVE 1 TO END-FILE.
144     PERFORM READ-CLI UNTIL (END-FILE = 0).
145
146 READ-CLI.
147     READ CLIENT NEXT
148     AT END MOVE 0 TO END-FILE
149     NOT AT END
150         DISPLAY CLI-CODE
151         DISPLAY CLI-SIGNAL
152         DISPLAY CLI-HISTORIQUE.
153
154 NOUV-STK.
155     DISPLAY "NOUVEAU STOCK".
156     DISPLAY "NUM PRODUIT : " WITH NO ADVANCING.
157     ACCEPT STK-CODE.
158
159     DISPLAY "LIBELLE : " WITH NO ADVANCING.
160     ACCEPT STK-LIBELLE.
161
162     DISPLAY "NIVEAU : " WITH NO ADVANCING.
163     ACCEPT STK-NIVEAU.
164
165     WRITE STK INVALID KEY DISPLAY "ERREUR ".
166
167 LIST-STK.
168     DISPLAY "LISTE DU STOCK ".
169
170     CLOSE STOCK.
171     OPEN I-O STOCK.
172
173     MOVE 1 TO END-FILE.
174     PERFORM READ-STK UNTIL END-FILE = 0.
175
176 READ-STK.
177     READ STOCK NEXT
178     AT END MOVE 0 TO END-FILE
179     NOT AT END
180         DISPLAY STK-CODE
181         DISPLAY STK-LIBELLE
182         DISPLAY STK-NIVEAU.
183
184 NOUV-COM.
185     DISPLAY "NOUVELLE COMMANDE".
186     DISPLAY "NUM COMMANDE : " WITH NO ADVANCING.
187     ACCEPT COM-CODE.
188
189     MOVE 1 TO END-FILE.
190     PERFORM READ-COM-CLI UNTIL END-FILE = 0.
191     MOVE CLI-SIGNAL TO SIGNALETIQUE.
192     DISPLAY NOM.
193     MOVE CLI-CODE TO COM-CLIENT.
194     MOVE CLI-HISTORIQUE TO LIST-ACHAT.

```

```

192
193 SET IND-DET TO 1.
194 MOVE 1 TO END-FILE.
195 PERFORM READ-DETAIL UNTIL END-FILE=0 OR IND-DET=21.
196 MOVE LIST-DETAIL TO COM-DETAIL.
197
198 WRITE COM INVALID KEY DISPLAY "ERREUR".
199
200 MOVE LIST-ACHAT TO CLI-HISTORIQUE.
201 REWRITE CLI INVALID KEY DISPLAY "ERREUR CLI".
202
203 READ-CODE-CLI.
204 DISPLAY "NUM DU CLIENT : " WITH NO ADVANCING.
205 ACCEPT CLI-CODE.
206 MOVE 0 TO END-FILE.
207 READ CLIENT INVALID KEY
208     DISPLAY "CLIENT INEXISTANT"
209     MOVE 1 TO END-FILE
210 END-READ.
211
212 READ-DETAIL.
213 DISPLAY "CODE DU PRODUIT (0 = FIN) : ".
214 ACCEPT CODE-PROD.
215 IF CODE-PROD = "0"
216     MOVE 0 TO END-FILE
217     MOVE 0 TO REF-STOCK-DE(IND-DET)
218 ELSE
219     PERFORM READ-CODE-PROD.
220
221 READ-CODE-PROD.
222 MOVE 1 TO EXIST-PROD.
223 MOVE CODE-PROD TO STK-CODE.
224 READ STOCK INVALID KEY
225 MOVE 0 TO EXIST-PROD.
226 IF EXIST-PROD = 0
227     DISPLAY "PRODUIT INEXISTANT"
228 ELSE
229     PERFORM UPDATE-COM-DETAIL.
230
231 UPDATE-COM-DETAIL.
232 MOVE 1 TO NEXT-DET.
233 DISPLAY "QTE COMMANDEE : " WITH NO ADVANCING.
234 ACCEPT Q-COM(IND-DET).
235 PERFORM UNTIL
236     REF-STOCK-DE(NEXT-DET) = CODE-PROD
237     OR IND-DET = NEXT-DET
238     ADD 1 TO NEXT-DET
239 END-PERFORM.
240 IF IND-DET = NEXT-DET
241     MOVE CODE-PROD TO REF-STOCK-DE(IND-DET)
242     PERFORM UPDATE-CLI-HISTO
243     SET IND-DET UP BY 1
244 ELSE
245     DISPLAY "ERREUR : PRODUIT DEJA COMMANDE".
246
247 UPDATE-CLI-HISTO.
248 SET IND TO 1.
249 PERFORM UNTIL
250     REF-STOCK-AC(IND) = CODE-PROD
251     OR REF-STOCK-AC(IND) = 0 OR IND = 101
252     SET IND UP BY 1
253 END-PERFORM.
254 IF IND = 101
255     DISPLAY "ERREUR : HISTORIQUE TROP PETIT"
256 EXIT.
257 IF REF-STOCK-AC(IND) = CODE-PROD
258     ADD Q-COM(IND-DET) TO TOT(IND)
259 ELSE
260     MOVE CODE-PROD TO REF-STOCK-AC(IND)
261     MOVE Q-COM(IND-DET) TO TOT(IND).
262
263 LIST-COM.
264 DISPLAY "LISTE DES COMMANDES ".
265 CLOSE COMMANDE.
266 OPEN I-O COMMANDE.
267 MOVE 1 TO END-FILE.
268 PERFORM READ-COM UNTIL END-FILE = 0.

```

```

269
270 READ-COM.
271 READ COMMANDE NEXT
272     AT END MOVE 0 TO END-FILE
273     NOT AT END
274         DISPLAY "COM-CODE " WITH NO ADVANCING
275         DISPLAY COM-CODE
276         DISPLAY "COM-CLIENT " WITH NO ADVANCING
277         DISPLAY COM-CLIENT
278         DISPLAY "COM-DETAIL "
279         MOVE COM-DETAIL TO LIST-DETAIL
280         SET IND-DET TO 1
281         MOVE 1 TO END-DETAIL
282         MOVE 0 TO IND-DET.
283         PERFORM DISPLAY-DETAIL UNTIL END-DETAIL=0.
284
285
286
287 INIT-HISTO.
288 SET IND TO 1.
289 PERFORM UNTIL IND = 100
290     MOVE 0 TO REF-STOCK-AC(IND)
291     MOVE 0 TO TOT(IND)
292     SET IND UP BY 1
293 END-PERFORM
294 MOVE LIST-ACHAT TO CLI-HISTORIQUE.
295
296 DISPLAY-DETAIL.
297 IF IND-DET = 21
298     MOVE 0 TO END-DETAIL
299     EXIT.
300 IF REF-STOCK-DE(IND-DET) = 0
301     MOVE 0 TO END-DETAIL
302 ELSE
303     DISPLAY REF-STOCK-DE(IND-DET)
304     DISPLAY Q-COM(IND-DET)
305     SET IND-DET UP BY 1.
306 MOVE IND-DET TO IND.
307 DISPLAY IND.

```

